A SIMPLE POLYNOMIAL GROUNDNESS ANALYSIS FOR LOGIC PROGRAMS

ANDY HEATON, MUHAMED ABO-ZAED,
MICHAEL CODISH AND ANDY KING

▷ The domain of positive Boolean functions, *Pos*, is by now well established for the analysis of the variable dependencies that arise within logic programs. Analyses based on *Pos* that use binary decision diagrams have been shown to be efficient for a wide range of practical programs. However, independent of the representation, a *Pos* analysis can never come with any efficiency guarantees because of its potential exponential behaviour. This paper considers groundness analysis based on a simple subdomain of *Pos* and compares its precision with that of *Pos*. ◁

1. Introduction

Many analyses for logic programs use Boolean functions to express dependencies between program variables. The most commonly used are the class of positive propositional formulae and its subclass of definite formulae, denoted *Pos* and *Def* respectively [1]. Numerous independent implementations have indicated that program analyses based on *Pos* and *Def* formulae are accurate and well suited to the analysis of logic programs, concurrent logic programs, constraint logic programs and deductive databases [1, 2, 6, 7, 9, 10, 11, 14, 21]. One of the best known applications of this type of analysis involves reasoning about groundness dependencies

Address correspondence to Andy Heaton, School of Computer Studies, University of Leeds, LS2 9JT, UK, heaton@scs.leeds.ac.uk, or Muhamed Abo-Zaed, Department of Mathematics and Computer Science, Ben-Gurion University, PoB 653, Beer-Sheba, Israel, abozaed@cs.bgu.ac.il, or Michael Codish, Department of Computer Science and Software Engineering, The University of Melbourne, Parkville 3052, Australia, mcodish@cs.bgu.ac.il, or Andy King, Computing Laboratory, University of Kent at Canterbury, Canterbury, CT2 7NF, UK, a.m.king@ukc.ac.uk.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/93/$3.50

in logic programs. For example, in this context the definite formula $x \wedge (y \leftarrow z)$ is interpreted to describe a program state in which x is definitely bound to a ground term and there exists a grounding dependency such that whenever z becomes bound to a ground term then so does y.

A variety of techniques have been devised to represent, maintain and manipulate various classes of propositional formulae for program analyses. Worth mentioning are: (1) the use of binary decision diagrams (BDD's) and their variants, such as reduced order binary decision diagrams (ROBDD's) [2, 7, 12, 20], which lead to fast Pos analyses for many benchmarks; (2) the use of dual Blake canonical form (DBCF) for Def [1]; and (3) the use of a set of possibly non-ground atoms over the alphabet $\{true, false\}$ to represent the truth table of a formula [6]. The first two techniques give very compact representations of Boolean functions with efficient, although complex, operations. The third technique is naive – its main attraction is the simplicity in which it can be implemented.

None of these techniques, however, come with any efficiency guarantees. This is not surprising considering the underlying complexity of the abstract domains. Each iteration of an analysis involves a test for equivalence to determine if a fixed point has been reached. For Pos, the equivalence problem is co-NP complete [1, Section 4]. Hence, for all practical purposes either the representation will be of exponential size or the test for equivalence will take exponential time. Moreover, both Def and Pos contain chains which are exponential in the arity of the predicates being analysed. A series of pathological input programs for Pos based groundness analysis are presented in [5]. These include programs for which the worst-case exponential number of iterations is encountered (for any representation) as well as programs for which exponentially large BDD's are generated. Furthermore, Pos analysis of some real benchmarks, like the Aquarius Prolog compiler, is problematic without (a space) widening because of high arity predicates that arise from extended definite clause grammars [12, 13]. In short, although the empirical evaluation results for Pos and Def presented in [1, 2, 6, 12, 21] suggest that analyses based on Pos and Def are efficient for many programs, both Pos and Def can never come with any performance guarantees and, occasionally, their performance is unacceptable. In more pragmatic terms, the problem is that a cautious compiler vendor is unlikely to adopt an analysis unless it comes with scalability guarantees.

In this paper our aim is to accelerate groundness analysis without losing too much precision in practice. Our approach is to restrain the size of the representation of the Pos formulae so as to put a reasonable bound both on the maximum number of iterations in an analysis as well as on the cost of a test for equivalence. We consider groundness analysis based on a linear subdomain of Pos. Namely, a domain in which the longest chain is linear (in the number of variables), and the size of a (call or answer pattern) description is linear (in the number of variables). Also the domain can be implemented with polynomial time complexity in the size of the input program. The subdomain of Pos which we consider consists of conjunctions of variables and equivalences between variables.

As an implementation vehicle we apply the simple technique described in [6] for Pos. In each iteration of the analysis (an abstraction of) each clause body is solved, in all possible ways, to infer a new description of the clause head. In our domain, each predicate is described by a single Prolog fact, so the evaluation is deterministic. This results in an analysis that scales smoothly.

2. Groundness Analysis Based on *Pos*

We follow the convention of identifying a truth assignment with the set of variables that it maps to *true*. For brevity, we introduce the map $model_X(f) = \{M \subseteq X \mid \phi_M(f) = true\}$ where ϕ_M is the truth assignment $\phi_M = \{x \mapsto true \mid x \in M\} \cup \{x \mapsto false \mid x \in X \setminus M\}$. For example, if $X = \{x,y\}$ then $model_X(x \wedge y) = \{\{x,y\}\}$ and $model_X(x \vee y) = \{\{x\},\{y\},\{x,y\}\}$. The set of positive Boolean functions over X is denoted Pos_X. Recall that a Boolean function f is positive if $X \in model_X(f)$. Hence $x \wedge y$, $x \vee y \in Pos_X$ but $\neg x \notin Pos_X$. The complete lattice $Pos_X^{\perp} = Pos_X \cup \{false\}$ is ordered by entailment $\models$ which is itself defined by $f_1 \models f_2$ if and only if $model_X(f_1) \subseteq model_X(f_2)$ where $f_1, f_2 \in Pos_X^{\perp}$. The bottom and top elements of $Pos_X^{\perp}$ are $false$ and $true$. A chain is a set in which no pairs of elements are incomparable. The abstract domain Pos_X is formally defined in [8, 17]. The interested reader can find additional details on analyses with Pos_X in [1, 2, 6, 7, 8, 10, 17, 21]. Let us recall briefly the implementation technique for groundness dependencies that is described in [6]. In this approach, a given logic program is first abstracted in such a way that the concrete semantics of the abstract program corresponds to the required groundness analysis. Figure 1 illustrates a Prolog program which rotates the elements of a list and its corresponding abstraction for groundness dependencies. An atom of the form `iff`$(x,[y,z])$ specifies the formula $x \leftrightarrow y \wedge z$ with the intended interpretation that x is ground if and only if y and z are. Consequently, a unification of the form $x = [y|z]$ in the concrete program is replaced by `iff`$(x,[y,z])$ in the abstract program. Similarly, a unification of the form $x = [\ ]$ in the concrete program can simply be replaced by `iff`$(x,[\ ])$ which specifies that x is definitely bound to a ground term. For the purpose of this paper it is sufficient to understand that the problem of analyzing the concrete program (on the left part of Figure 1) is reduced to the problem of computing the concrete semantics of the abstract program (in the middle and on the right). For additional details of why this is so, refer to [6, 8, 17].

Concrete `rotate`	*Abstract* `rotate`	*Auxiliary predicates*
`rotate(`xs`,`ys`) :-`	`rotate(`xs`,`ys`) :-`	`iff(true,[ ]).`
`append(`as`,`bs`,`xs`),`	`append(`as`,`bs`,`xs`),`	`iff(true,[true|`xs`]) :-`
`append(`bs`,`as`,`ys`).`	`append(`bs`,`as`,`ys`).`	`iff(true,`xs`).`
		`iff(false,`xs`) :-`
`append(`xs`,`ys`,`zs`) :-`	`append(`xs`,`ys`,`zs`) :-`	`member(false,`xs`).`
$xs = [\ ]$,	`iff(`xs`,[ ]),`	
$ys = zs$.	`iff(`ys`,[`zs`]).`	
`append(`xs`,`ys`,`zs`) :-`	`append(`xs`,`ys`,`zs`) :-`	
$xs = [x|xs1]$,	`iff(`xs`,[`$x, xs1$`]),`	
$zs = [x|zs1]$,	`iff(`zs`,[`$x, zs1$`]),`	
`append(`$xs1$`,`ys`,`$zs1$`).`	`append(`$xs1$`,`ys`,`$zs1$`).`	

FIGURE 1. Corresponding concrete and abstract programs.

The concrete (bottom-up) semantics of a logic program (with a finite minimal model) is easily computed using simple meta-interpreters such as those described in [4, 6]. The basic idea is to perform iterations in which we solve clause bodies using the facts derived so far to derive new instances of clause heads. The last iteration is the first one in which no new head instances are derived. Applying this approach to the abstract rotate program from Figure 1 gives the following atoms:

```
rotate(x,x).     append(true,true,true).
                 append(false,y,false).
                 append(x,false,false).
```

which are interpreted as representing the propositional formula $x_1 \leftrightarrow x_2$ and $(x_1 \wedge x_2) \leftrightarrow x_3$ for the atoms $\mathtt{rotate}(x_1, x_2)$ and $\mathtt{append}(x_1, x_2, x_3)$ respectively. This illustrates a goal-independent analysis. Goal-dependent analyses are supported by applying Magic sets or similar techniques (see e.g. [6]). The simple, naive scheme described above provides the basis for various more efficient implementations based on semi-naive evaluation, strongly connected components, and other optimisation techniques. For further details and examples of meta-interpreters in Prolog which perform this type of evaluation see [4, 6].

The inefficiency of the above strategy stems both from the representation, as well as from the evaluation mechanism. Consider a single clause $h \leftarrow b_1, \ldots, b_n$ in an iteration of the evaluation. Each call in the body might match a number of atoms (corresponding to disjuncts in a disjunctive normal form) which is exponential in the arity of the call. Solving the clause body in all possible ways is thus also exponential.

Sophisticated techniques such as those based on BDD's [1] give a more compact representation of Boolean functions although this is achieved at the cost of more complex operations. However, as we have noted, for all practical purposes any *Pos* based analysis is inherently exponential.

3. Simplifying *Pos*

We propose a simple subdomain of *Pos* for which the representation of formulae is linear in the size of the program, as is the maximum number of iterations in an analysis. Formulae in this domain consist of conjunctions of variables and equivalences between variables. For example, $(x_1 \leftrightarrow x_2) \wedge x_3 \wedge (x_1 \leftrightarrow x_4)$. We call this class of formulae *EPos*. Like *Pos*, *EPos* is ordered by $\models$. It is interesting to note that *EPos* is only slightly richer than its subdomain *Con* [15, 18] which consists of conjunctions of variables. However, as we shall see, *EPos* gives much greater precision than *Con* for groundness analysis of logic programs. Moreover, it shares with *Con* the important property that its longest chain has linear length. The proof of this result relies on the observation that an *EPos* formula over X can be represented as a non-ground tuple. For example, if $X = \{x_1, ..., x_5\}$, then $t = \langle x_1, true, x_1, x_4, true \rangle$ represents $(x_1 \leftrightarrow x_3) \wedge x_2 \wedge x_5$. Similarly, $\langle true, x_2, x_3, x_2, x_2 \rangle$ encodes $x_1 \wedge (x_2 \leftrightarrow x_4) \wedge (x_4 \leftrightarrow x_5)$.

Lemma 3.1. Let X be a set of n variables. The longest chain in the lattice $EPos_X^{\perp} = EPos_X \cup \{false\}$ has length $n + 2$.

PROOF. Let $X = \{x_1, x_2, \ldots, x_n\}$, then any function f in $EPos_X$ can be represented as an n-tuple t such that f has 2^d models where d is the number of distinct variables in t. For example with $n = 5$, if $f = (x_1 \leftrightarrow x_3) \wedge x_2 \wedge x_5$, then f is encoded by $\langle x_1, true, x_1, x_4, true \rangle$ which has 2 distinct variables and $model_X(f) = \{\{x_2, x_5\}, \{x_1, x_2, x_3, x_5\}, \{x_2, x_4, x_5\}, \{x_1, x_2, x_3, x_4, x_5\}\}$. If $f \models f'$ and $f' \not\models f$, then $models_X(f) \subset models_X(f')$ and hence f' has more models than f. This implies that the tuple representation of f' contains more distinct variables than that of f. Since d is bounded by n, a chain in $EPos_X$ can, at most, contain $n+1$ functions so that the length of a chain in $EPos_X^\perp$ is bounded by $n+2$. A chain of length $n+2$ is obtained as $false$, $x_1 \wedge x_2 \wedge \ldots \wedge x_n$, $x_2 \wedge x_3 \wedge \ldots \wedge x_n$, $\cdots$, x_n, $true$.

As a consequence, we obtain the following theorem:

Theorem 3.1. The number of iterations required to obtain a fixpoint in a (univariant) analysis based on the $EPos$ domain is linear in the sum of the arities of the predicate symbols that occur in the program.

Note that a univariant analysis maintains one success pattern (or one call and answer pattern) per predicate [22].

PROOF. Suppose a program contains predicates $p_1, \ldots, p_n$ of arity $a_1, \ldots, a_n$. The database of success patterns is a tuple $\langle f_1, \ldots, f_n \rangle$ of formulae for $p_1, \ldots, p_n$ respectively. The ordering for tuples is standard and defined as follows: $\langle f_1, \ldots, f_n \rangle \models \langle f_1', \ldots, f_n' \rangle$ iff $f_1 \models f_1', \ldots, f_n \models f_n$. Therefore the length of the longest chain of $EPos$ tuples is $\sum_{i=1}^{n}(a_i + 2)$ and hence, a univariant $EPos$ analysis will take a linear number of iterations to reach the fixpoint. Similarly, the number of iterations for an analysis deriving call and answer patterns is no more than $2 \cdot \sum_{i=1}^{n}(a_i + 2)$.

The fact that a formula can be represented by a non-ground tuple turns out to be important for adapting the implementation technique of [6]. It enables, for example, the call pattern pair $\langle a, f \rangle$ where $a = p(x_1, x_2, x_3, x_4, x_5)$ and $f = (x_1 \leftrightarrow x_2) \wedge x_3 \wedge (x_1 \leftrightarrow x_4)$ to be succinctly represented by a single non-ground atom, namely, $p(x_1, x_1, true, x_1, x_5)$. We consider three analysis strategies for adapting the implementation technique of [6] to $EPos$. These differ in the way that the `iff/2` atoms are handled:

Non-deterministic `iff/2` *atoms:* The first strategy is based on the technique described in [6]. Each user defined predicate is represented as an $EPos$ formula by a single non-ground atom. The `iff/2` atoms, however, are described in Pos using the `iff/2` (auxiliary) predicate of Figure 1. This analysis turns out to lose little precision in comparison with Pos. As a consequence of the non-determinism in the `iff/2` atoms, each clause in the program may have many solutions. These solutions are combined by a least upper bound operation in $EPos$. Specifically, a new (Pos) description is combined with an existing ($EPos$) description by computing the most general subsumer of the two. The non-determinism in this approach is a source of inefficiency. In the following we refer to this strategy as $EPos_N$ (N stands for nondeterministic).

Deterministic `iff/2` *atoms:* The second approach avoids the inefficiency of the first by solving the `iff/2` atoms deterministically. Namely, only when they have a single solution. This can be achieved using code such as that given in Figure 2 (for 4 variables). The first clause in Figure 2 is for the case when w has the value $true$. Then x, y and z are all instantiated to $true$. The next three clauses are for the cases when w does not have the value $true$, but at least two of x, y, z do. Aliasing information can be deduced in these cases. The final clause is for the case where no deterministic information can be deduced. The `iff/2` atom is ignored in this instance. This type of analysis turns out to be very efficient as the entire clause body is solved deterministically. However, ignoring the non-deterministic `iff/2` atoms is a source of imprecision. In the following we refer to this strategy as $EPos_D$ (D is for deterministic).

Deterministic `iff/2` *atoms with local iteration:* The third strategy is intended as a compromise between the precision of the first and the efficiency of the second. It is based on the observation that solving the `iff/2` atoms in the right order can reduce the number of ignored atoms, and thus improve precision.

The analyser iterates over the `iff/2` atoms in the body of a clause checking to see whether any `iff/2` atoms that were not deterministic when they were first encountered, can now be solved deterministically. Applying this scheme, the evaluation of a clause body remains fully deterministic. In addition, we also check to see if a clause body contains pairs of atoms of the form `iff(`x`,`Y`), iff(`y`,`Y`)` in which x, y and (the list of variables) Y are not instantiated to ground terms. Here it can be inferred that $x = y$. In the end, any remaining non-deterministic `iff/2` atoms are ignored. Of course the first technique may still be more precise since there may still be collections of non-deterministic `iff/2` atoms that contain information (about the clause head) expressible in $EPos$. For instance if a clause body contains the atoms `iff(`x`,`$[w,y,z]$`), iff(`y`,`$[w,z]$`)` in which w, x, y, z are not instantiated, the third approach will not detect that these could have been replaced by a single call of the form $x = y$ and this information will be lost. An $EPos$ analysis that applies this third method is called $EPos_L$ (L stands for local iteration).

```
iff(w, [x, y, z]) :- w == true, !, x = true, y = true, z = true.
iff(w, [x, y, z]) :- x == true, y == true, !, w = z.
iff(w, [x, y, z]) :- x == true, z == true, !, w = y.
iff(w, [x, y, z]) :- y == true, z == true, !, w = x.
iff(_,_).
```

FIGURE 2. Deterministically solving `iff(`$w, [x, y, z]$`)`.

Theorem 3.2. Analysis using $EPos_L$ has polynomial time complexity in the space required to store the input program.

PROOF. Note that the space required to store a program P is a bound for: the sum of the arities of the predicates in P, denoted a_P; the maximal number of variables in a clause of P, denoted v_P; and the maximal number of atoms in the body of a clause of P, denoted b_P. By Theorem 3.1, the number of iterations in a $EPos_L$ analysis is linear in a_P. Hence, it is sufficient to show that each of the following operations have polynomial complexity bounds:

Rename: Let f and f' be $EPos$ formulae equivalent up to renaming of variables. Then a k-tuple encoding f can be renamed to a k-tuple encoding f' in $O(k \log(k))$ steps. Note that $k \leq v_P$.

Join: The most general subsumer of two k-tuples that encode $EPos$ formulae, can be computed in $O(k\log(k))$ time using Plotkin's anti-unification algorithm [19]. Observe that $k \leq a_P$.

Equivalence check: The test for equivalence of $EPos$ formulae over k variables can be implemented in $O(k)$ simple unifications (that do not involve compound terms). Let $f, f' \in EPos$ be represented as tuples $\langle t_1, \ldots, t_k\rangle$ and $\langle t_1{}', \ldots, t_k{}'\rangle$. The equivalence check amounts to two tests: $f \models f'$ and $f' \models f$. Consider the test $f \models f'$ which can be implemented as follows: the terms in $t_1, \ldots, t_k$ are considered in order from 1 to k. If t_1 is a variable it is assigned the value 1, if t_2 is a variable it is assigned the value 2 and so on. After k steps the tuple representation of f is a ground tuple $\bar{t}$. For example, if f is represented as $\langle x_1, true, x_1, x_4, true\rangle$ then $\bar{t} = \langle 1, true, 1, 4, true\rangle$. Now, if $\bar{t}$ and $\langle t_1{}', \ldots, t_k{}'\rangle$ are unifiable and this can be checked with k simple unifications (that do not involve compound terms), then $f \models f'$. Note that $k \leq a_P$.

Meet: Consider the meet operations to solve the body of a single (abstract) clause of P consisting of $r \leq v_P$ variables and $s \leq b_P$ body atoms. First, consider the `iff/2` atoms in the clause body. It takes $O(k^2)$ steps to check if an atom of the form `iff(`x`,[`$y_1, \ldots, y_k$`])` ($k \leq r$) can be solved deterministically and the evaluation strategy will reconsider an `iff/2` atom at most s times. Second, consider a call to a user-defined predicate p/k in the body of the clause. It takes k ($k \leq r$) simple unifications (that do not involve compound terms) to match the call with its current call or answer pattern.

4. Experimental Results

This section presents an experimental evaluation. We focus on the goal-dependent groundness analysis of 76 Prolog and CLP($\mathcal{R}$) programs ranging in size from 2 to over 4000 clauses. We summarise by saying:

- $EPos_N$ is as precise as Pos on 62 of our 76 benchmark programs; and loses less than 10% of the (ground argument) information on 73 programs.
- $EPos_L$ is as precise as $EPos_N$ and scales well.

Our experiments are based on an analyser coded in Prolog using induced Magic-sets [4] and eager evaluation [23] to perform efficient goal-dependent bottom-up evaluation. Induced magic is an interpreter based implementation technique that avoids the transformation associated with Magic-sets (calls and answers are expressed directly within the interpreter) and also avoids much of the re-computation that can arise in magic clause bodies. Eager evaluation involves an "almost semi-naive" strategy which whenever a new head atom is derived invokes a recursive procedure to ensure that every clause that has that atom in its body is re-evaluated. One advantage is that there is no overhead in distinguishing old and new atoms. The benchmarks and a simplified version of the analyser can be obtained from http://www.cs.ukc.ac.uk/people/staff/amk. This distribution includes documentation on how builtins and constraints are abstracted (since our precision results may deviate from those reported by others if they abstract programs differently).

Tables 1 and 2 summarise the analysis times and precision results obtained for $EPos_N$ and $EPos_L$ together with those for Con and Pos. The benchmark programs are ordered according to the number of (distinct abstract) clauses they contain. The Pos analyser is built on a ROBDD package coded by Armstrong and Schachte [1]. The Con and Pos experiments are performed mainly for comparing precision against $EPos_N$ and $EPos_L$, and therefore the Pos analysis is not widened [5, 12]. The pre-processing times are included in the abs column. The fixpoint columns give the time, in seconds, to compute the fixpoint for each of the four techniques. C, L, N and P abbreviate Con, $EPos_L$, $EPos_N$ and Pos, respectively. The precision columns give the total number of ground arguments in the call and answer patterns (and exclude those ground arguments for predicates introduced by normalising the program into definite clauses). The % prec columns express the loss of precision relative to Pos. The analyser is coded in SICStus 3.5 and the experiments performed on a 270MHz Sun Ultra 5 with 128 MByte of RAM running Solaris 2.6.

The precision results for the Con analysis reconfirm that this domain is not expressive enough: Con loses precision wrt Pos on 49 of the 76 programs, and for 16 of these Con loses 50% or more of the ground arguments inferred by Pos. In contrast, $EPos_N$ loses precision on 14 programs, and from these it loses more than 10% of the groundness information on 3 benchmarks. For the programs where $EPos_N$ loses precision wrt to Pos, it is interesting to see the precision obtained with other subdomains of Pos. The subdomains we consider are: (1) Def; (2) conjunctions of variables and implications of the form $x \leftarrow y$ – a class of formulae that we shall label Imp. We have implemented Def and Imp (abbreviated to D and I in Table 4) analyses by restricting the call and answer patterns in our Pos analyser to Def and Imp formulae, respectively. From Table 4 we conclude that $EPos_N$ loses precision for append.pl, nandc.pl, mastermind.pl, lnprolog.pl, chat_parser.pl, essln.pl, chat_80.pl and aqua_c.pl because it cannot track implications of the form $x \leftarrow y$ across procedure boundaries. Precision is lost for rotate.pl, ime_v2-2-1.pl, neural.pl, press.pl, rubik.pl, lnprolog.pl and aqua_c.pl because $EPos_N$ cannot adequately trace implications such as $x \leftarrow \wedge_{i=1}^{n} x_i$ where $n \geq 2$. Finally, precision is lost for rotate.pl and sim_v5-2.pl because $EPos_N$ cannot capture disjunctive dependencies.

The use of $EPos_L$ instead of $EPos_N$ is justified when observing that the same precision results are obtained for all programs. Moreover, the timings for $EPos_L$ are sometimes considerably faster than those for $EPos_N$ with all but 3 programs giving fixpoint times under one tenth of a second and the slowest analysis taking 1.83 seconds. Disabling local iteration, to obtain $EPos_D$, has a disastrous impact on precision collapsing the accuracy to near that of Con. We also conducted experiments on goal-independent analysis. $EPos_L$ loses precision on 21 of the 76 programs and for 8 of these the loss is more than 10% of the groundness arguments. A complete set of goal-independent analysis times and precision results can be found in the distribution.

For completeness, Table 4 gives a comparison of our $EPos_L$ analysis against two of the fastest Pos analysers that are described in the literature. The Pos analysis of [12] is implemented in the SML-based GENA framework on a Sun 20 with 64 Mbytes of memory. This analysis widens large BDDs by projecting formulae for large arity predicates onto Con. This shows up in the analysis time for aqua_c.pl. The domain operations of the Pos analysis of [20] are coded in C while the framework itself is

			fixpoint time				precision				% prec		
file	size	abs	C	L	N	P	C	L	N	P	C	L	N
append.pl	2	0.01	0.01	0.00	0.01	0.01	3	3	3	4	25	25	25
rotate.pl	3	0.01	0.00	0.01	0.00	0.00	2	2	2	6	67	67	67
mortgage.clpr	4	0.01	0.00	0.00	0.03	0.00	6	6	6	6	0	0	0
qsort.pl	6	0.01	0.00	0.00	0.00	0.01	10	11	11	11	9	0	0
rev.pl	6	0.00	0.01	0.00	0.00	0.01	0	0	0	0	0	0	0
queens.pl	9	0.00	0.00	0.00	0.00	0.01	2	3	3	3	33	0	0
zebra.pl	9	0.01	0.00	0.00	0.01	0.02	11	19	19	19	42	0	0
laplace.clpr	11	0.02	0.00	0.00	0.01	0.01	0	0	0	0	0	0	0
shape.pl	11	0.01	0.00	0.01	0.02	0.02	3	6	6	6	50	0	0
parity.pl	12	0.01	0.01	0.01	0.01	4651.95	0	0	0	0	0	0	0
treeorder.pl	12	0.00	0.00	0.00	0.01	0.01	0	0	0	0	0	0	0
fastcolor.pl	13	0.02	0.00	0.01	0.00	0.01	14	14	14	14	0	0	0
music.pl	13	0.01	0.00	0.02	6.02	0.03	2	2	2	2	0	0	0
serialize.pl	13	0.01	0.00	0.01	0.02	0.02	3	3	3	3	0	0	0
crypt_wamcc.pl	19	0.01	0.00	0.01	0.00	0.04	26	31	31	31	16	0	0
option.clpr	19	0.01	0.00	0.00	0.03	0.03	42	42	42	42	0	0	0
circuit.clpr	20	0.01	0.01	0.01	0.71	0.02	3	3	3	3	0	0	0
air.clpr	20	0.01	0.00	0.01	0.10	0.03	9	9	9	9	0	0	0
dnf.clpr	23	0.02	0.00	0.01	0.00	0.01	0	8	8	8	100	0	0
dcg.pl	23	0.02	0.00	0.01	0.00	0.01	54	59	59	59	8	0	0
hamiltonian.pl	23	0.01	0.00	0.00	0.01	0.01	36	37	37	37	3	0	0
poly10.pl	29	0.02	0.00	0.00	0.00	0.01	0	0	0	0	0	0	0
semi.pl	31	0.02	0.01	0.02	1.26	0.29	23	28	28	28	18	0	0
life.pl	32	0.02	0.00	0.00	0.01	0.02	52	58	58	58	10	0	0
ronp.pl	34	0.02	0.00	0.00	0.02	0.03	10	10	10	10	0	0	0
rings-on-pegs.clpr	34	0.02	0.01	0.01	0.04	0.04	11	11	11	11	0	0	0
meta.pl	35	0.02	0.01	0.00	0.01	0.02	0	1	1	1	100	0	0
browse.pl	36	0.02	0.01	0.01	0.01	0.02	41	41	41	41	0	0	0
gabriel.pl	38	0.02	0.00	0.00	0.02	0.03	37	37	37	37	0	0	0
tsp.pl	38	0.03	0.01	0.01	0.01	0.05	82	122	122	122	33	0	0
nandc.pl	40	0.03	0.01	0.01	0.02	0.03	13	34	34	37	65	8	8
csg.clpr	48	0.04	0.01	0.01	$>$ 172800	0.07	12	12	–	12	0	0	–
disj_r.pl	48	0.02	0.01	0.01	0.01	0.04	38	97	97	97	61	0	0
ga.pl	48	0.06	0.01	0.00	0.01	0.04	127	141	141	141	10	0	0
critical.clpr	49	0.03	0.01	0.01	14462.05	0.04	14	14	14	14	0	0	0
scc1.pl	52	0.04	0.01	0.00	0.02	0.12	44	90	90	90	51	0	0
mastermind.pl	59	0.04	0.01	0.01	0.02	0.10	10	43	43	44	77	2	2
ime_v2-2-1.pl	53	0.03	0.01	0.02	0.02	0.09	77	100	100	101	24	1	1
robot.pl	53	0.04	0.01	0.01	0.00	0.02	9	41	41	41	78	0	0
cs_r.pl	54	0.05	0.01	0.01	0.02	0.04	36	149	149	149	76	0	0
tictactoe.pl	56	0.05	0.01	0.01	0.01	0.04	55	60	60	60	8	0	0
flatten.pl	56	0.04	0.01	0.02	0.05	0.09	26	27	27	27	4	0	0
dialog.pl	61	0.03	0.01	0.01	0.01	0.03	46	70	70	70	34	0	0
map.pl	66	0.02	0.02	0.01	0.01	0.07	17	17	17	17	0	0	0
neural.pl	67	0.05	0.02	0.01	0.02	0.06	85	121	121	123	31	2	2
bridge.clpr	69	0.08	0.00	0.01	0.02	0.02	9	9	9	9	0	0	0
conman.pl	71	0.04	0.00	0.00	0.00	0.02	6	6	6	6	0	0	0
kalah.pl	78	0.04	0.02	0.01	0.01	0.05	91	199	199	199	54	0	0
unify.pl	79	0.04	0.01	0.01	0.06	0.09	69	70	70	70	1	0	0
nbody.pl	85	0.08	0.02	0.03	0.07	0.11	87	113	113	113	23	0	0

TABLE 1. Smaller benchmarks: Precision and Times

file	size	abs	fixpoint time C	L	N	P	precision C	L	N	P	% prec C	L	N
peep.pl	88	0.09	0.01	0.02	0.03	0.06	8	10	10	10	20	0	0
boyer.pl	95	0.06	0.01	0.01	0.02	0.06	3	3	3	3	0	0	0
bryant.pl	95	0.07	0.02	0.04	0.22	0.15	94	99	99	99	5	0	0
sdda.pl	99	0.05	0.01	0.02	0.06	0.07	17	17	17	17	0	0	0
read.pl	106	0.07	0.01	0.03	0.03	0.12	85	99	99	99	14	0	0
press.pl	109	0.07	0.03	0.04	0.10	0.18	45	52	52	53	15	2	2
qplan.pl	109	0.07	0.03	0.02	0.02	0.09	42	216	216	216	81	0	0
trs.pl	112	0.09	0.02	0.07	16626.17	0.68	13	13	13	13	0	0	0
reducer.pl	113	0.07	0.01	0.03	0.14	0.16	36	41	41	41	12	0	0
simple_analyzer.pl	140	0.08	0.02	0.03	0.11	0.48	88	89	89	89	1	0	0
dbqas.pl	146	0.09	0.01	0.01	0.05	0.06	36	43	43	43	16	0	0
ann.pl	148	0.15	0.02	0.05	0.18	0.69	71	71	71	71	0	0	0
asm.pl	175	0.13	0.03	0.03	0.04	0.14	89	90	90	90	1	0	0
nand.pl	181	0.11	0.18	0.03	0.05	0.20	123	402	402	402	69	0	0
rubik.pl	219	0.17	0.05	0.05	0.51	0.31	158	171	171	179	12	4	4
lnprolog.pl	221	0.10	0.03	0.05	0.07	0.14	54	110	110	143	62	23	23
ili.pl	225	0.12	0.03	0.05	0.27	0.25	4	4	4	4	0	0	0
sim.pl	250	0.18	0.04	0.07	1.22	0.75	81	100	100	100	19	0	0
strips.pl	261	0.16	0.01	0.01	0.02	0.14	85	142	142	142	40	0	0
chat_parser.pl	281	0.23	0.14	0.16	0.58	0.82	444	504	504	505	12	0	0
sim_v5-2.pl	288	0.16	0.07	0.04	0.03	0.20	80	455	455	457	82	0	0
peval.pl	329	0.16	0.02	0.04	0.13	0.30	28	28	28	28	0	0	0
aircraft.pl	397	0.45	1.76	0.09	0.15	0.57	228	687	687	687	67	0	0
essln.pl	605	0.35	0.08	0.11	0.37	1.20	126	174	174	178	29	2	2
chat_80.pl	888	0.83	0.29	0.40	2.79	3.39	471	852	852	855	45	0	0
aqua_c.pl	4024	3.55	1.72	1.83	440.08	358.83	1148	1227	1227	1290	11	5	5

TABLE 2. Larger benchmarks: Precision and Times

file	precision C	L	N	I	D	P	% precision C	L	N	I	D
append.pl	3	3	3	4	4	4	25	25	25	0	0
rotate.pl	2	2	2	2	3	6	67	67	67	67	50
nandc.pl	13	34	34	37	37	37	65	8	8	0	0
mastermind.pl	10	43	43	44	44	44	77	2	2	0	0
ime_v2-2-1.pl	77	100	100	100	101	101	24	1	1	1	0
neural.pl	85	121	121	121	123	123	31	2	2	2	0
press.pl	45	52	52	52	53	53	15	2	2	2	0
rubik.pl	158	171	171	171	179	179	12	4	4	4	0
lnprolog.pl	54	110	110	111	143	143	62	23	23	22	0
chat_parser.pl	444	504	504	505	505	505	12	0	0	0	0
sim_v5-2.pl	80	455	455	455	455	457	82	0	0	0	0
essln.pl	126	174	174	178	178	178	29	2	2	0	0
chat_80.pl	471	852	852	855	855	855	45	0	0	0	0
aqua_c.pl	1148	1227	1227	1228	1290	1290	12	5	5	5	0

TABLE 3. Additional precision results for Imp and Def

written in Prolog. It has been implemented and benchmarked on a Sun Ultra 5 with a 270 MHz Sun UltraSparc processor and 320 Mbytes of memory. To match the architecture of [12] as closely as possible, we have also timed our $EPos_L$ analyser on a Sun-20 with 64 MByte of memory that is equipped with a 50MHz processor. Precise processor details are not given in [12] and, in fact, his machine could be as much as two times as fast as ours. For ease of comparison, we repeat some of the timings for our Sun Ultra 5 experiments.

	Pos		$EPos_L$	
file	Sun 20 [12]	Ultra 5 [20]	Sun 20	Ultra 5
peep.pl	0.06	0.02	0.07	0.02
boyer.pl	0.10	0.01	0.07	0.01
read.pl	0.16	0.06	0.12	0.03
press.pl	0.26	0.03	0.22	0.04
simple_analyzer.pl		0.07	0.18	0.03
ann.pl	0.19	0.03	0.16	0.05
nand.pl	0.33	0.10	0.13	0.03
sim.pl		0.24	0.32	0.07
chat_parser.pl	1.48	0.25	0.71	0.16
sim_v5-2.pl		0.18	0.15	0.04
peval.pl		0.04	0.20	0.04
chat_80.pl	4.31	0.66	1.84	0.40
aqua_c.pl	29.82	86.15	8.40	1.83

TABLE 4. Performance comparison

5. Discussion

We have presented a simplification of the *Pos* domain consisting of conjunctions of variables and equivalences between variables. The implementation is based on the technique of [6] and applies a simple local iteration technique to maintain a deterministic evaluation when solving a clause body. The analyser comes with polynomial performance guarantees and the core analyser (the meta-interpreter) can be coded succinctly. In fact the main software development effort in implementing the $EPos_L$ analyser was in elaborating the abstracter module to handle builtins accurately and correctly.

For BDD based *Pos* analysers, widening is feasible and resulting analyses should offer improved accuracy over $EPos_L$ without incurring excessive run times. The operations of BDD's, however, require much greater coding effort than those of $EPos_L$. In principle, the *Pos* implementation of [6] can also be widened, trigger in SICStus, say, with a timeout predicate. In practice, however, timeouts do not fit well with the eager evaluation model of induced magic.

A limited form of polyvariance could be supported for, say libraries, by applying $EPos_L$ to each exported predicate with its anticipated call patterns (and a safe approximation *true*). This would give a set of call and answer pattern pairs. A suitable answer pattern for a call to an imported predicate could then be found by matching the call to the most accurate, safe call pattern in the set of pairs.

The complexity result given by Theorem 3.2 is unaffected provided there is a fixed bound to the number of call and answer pattern pairs permitted for each exported predicate.

Work relating to our approach is presented in [2, 20] where a hybrid representation for Pos is considered. This BDD based representation, named GER, consists of three components: a set of ground variables (G), a set of equivalent variables (E), and an ROBDD for more complex dependencies (R). This enables a significant speed up in analysis times as the information in the G and E components is used to reduce the size of the R component which in turn makes the ROBDD operations less expensive. The information in our domain corresponds precisely to that captured in the G and E components of this representation. Our analysis technique differs in that we omit the ROBDD component altogether. Instead we apply a simple local iteration technique. Note that the BDD based Pos analyser used in the experimental results section does not use a GER represention.

Our local iteration technique has similarities with re-execution [16], which in turn goes back to the repeat previous call strategy of [3]. However there are three important differences in our approach compared to re-execution: (1) the iteration involves only `iff/2` predicates; (2) the iteration is local to the context of a single (abstract) clause; and (3) each `iff/2` atom will be solved at most once. While iterating over the calls in a clause body, each call may be inspected more than once to check if the call is deterministic, but calls are never solved more than once. Hence, the precision of a $EPos_L$ analysis never exceeds that of the corresponding $EPos_N$ analysis. In practice, the $EPos_L$ analyser maintains the same precision as the $EPos_N$ analysis, but at a fraction of the cost. Also in [16], a mode analysis with a Pattern domain is compared against an analysis with a domain that traces ground, var and any modes, and also equivalences between variables.

Acknowledgements

We would like to thank Florence Benoy, Pat Hill and Cohavit Taboch for discussions, Peter Schachte for help with the ROBDD comparison and Roberto Bagnara and the other reviewers of this paper for their useful comments. This work was funded, in part, by the UK ESPRC Grants GR/MO5645 and GR/MO8769 and by the Israel Science Foundation. Much of the work was carried out while Andy Heaton was visiting Ben-Gurion University.

REFERENCES

1. T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two Classes of Boolean Functions for Dependency analysis. *Science of Computer Programming*, 31(1):3–45, 1998.
2. R. Bagnara and P. Schachte. Factorizing Equivalent Variable Pairs in ROBDD-Based Implementations of *Pos*. In *Proceedings of the Seventh International Conference on Algebraic Methodology and Software Technology*, pages 471–485. Springer-Verlag, 1999. LNCS 1548.
3. M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *The Journal of Logic Programming*, 10(2):91–124, 1991.
4. M. Codish. Efficient Goal Directed Bottom-up Evaluation of Logic Programs. *The Journal of Logic Programming*, 38(3):354–370, 1999.

5. M. Codish. Worst-Case Groundness Analysis using Positive Boolean Functions. *The Journal of Logic Programming*, 1999. (to appear).
6. M. Codish and B. Demoen. Analysing Logic Programs using "prop"-ositional Logic Programs and a Magic Wand. *The Journal of Logic Programming*, 25(3):249–274, 1995.
7. M.-M. Corsini, K. Musumbu, A. Rauzy, and B. Le Charlier. Efficient Bottom-up Abstract Interpretation of Prolog by means of Constraint Solving over Finite Domains. In *Programming Language Implementation and Logic Programming*, pages 75–91. Springer-Verlag, 1993. LNCS 714.
8. A. Cortesi, G. Filé, and W. Winsborough. Optimal Groundness Analysis using Propositional Logic. *The Journal of Logic Programming*, 27(2):137–168, 1996.
9. P. Dart. On Derived Dependencies and Connected Databases. *The Journal of Logic Programming*, 11(2):163–188, 1991.
10. S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical Program Analysis Using General Purpose Logic Programming Systems — A Case Study. In *Programming Language Design and Implementation*, pages 117–126. ACM Press, 1996.
11. S. Debray and D. S. Warren. Automatic Mode Inference for Logic Programs. *The Journal of Logic Programming*, 5:207–229, 1988.
12. C. Fecht. *Abstrakte Interpretation logischer Programme: Theorie, Implementierung, Generierung*. PhD thesis, Universität des Saarlandes, 1997.
13. C. Fecht. Personal communication on *Pos*, DCG's and large arity predicates. November 1997.
14. M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM TOPLAS*, 18(5):564–614, 1996.
15. N. Jones and H. Søndergaard. A Semantics-based Framework for the Abstract Interpretation of Prolog. In *Abstract Interpretation of Declarative Languages*, pages 123–142. Ellis Horwood Limited, 1987.
16. B. Le Charlier and P. Van Hentenryck. Reexecution in Abstract Interpretation of Prolog. *Acta Informatica*, 32:209–253, 1995.
17. K. Marriott and H. Søndergaard. Precise and Efficient Groundness Analysis for Logic Programs. *ACM Lett. Program. Lang. Syst.*, 2(4):181–196, 1993.
18. C. Mellish. Abstract Interpretation of Prolog Programs. In *Third International Conference on Logic Programming*, pages 463–474. Springer, 1986. LNCS 225.
19. G. Plotkin. A Note on Inductive Generalisation. *Machine Intelligence*, 5:153–163, 1970.
20. P. Schachte. *Precise and Efficient Static Analysis of Logic Programs*. PhD thesis, Department of Computer Science, The University of Melbourne, Melbourne, Australia, 1999.
21. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Evaluation of the Domain *Prop*. *The Journal of Logic Programming*, 23(3):237–278, 1995.
22. P. Van Hentenryck, O. Degimbe, B. Le Charlier, and L. Michel. The Impact of Granularity in Abstract Interpretation of Prolog. In *Proceedings of the Workshop on Static Analysis*, pages 1–14. Springer-Verlag, 1993. LNCS 724.
23. J. Wunderwald. Memoing Evaluation by Source-to-Source Transformation. In *Fifth International Workshop on Logic Program Synthesis and Transformation*, pages 17–32. Springer, 1995. LNCS 1048.